EUROPEAN COMMISSION

DIGIT
Connecting Europe Facility

Access Point

Domibus 4.1.3

Plugin Cookbook

Version [3.3]

Status [Final]

Date: 14/04/2020

Document Approver(s):

Approver Name	Role
Adrien FERIAL	Project Manager

Document Reviewers:

Reviewer Name	Role
Caroline AEBY and Chaouki BERRAH	Technical Writers
Cosmin BACIU	Technical Leader

Summary of Changes:

Version	Date	Created by	Short Description of Changes
0.01	21.06.2016	Christian KOCH	Initial version
0.02	14.07.2016	Christian KOCH	Changes according to comments
0.03	21.07.2016	Christian KOCH	Added link to administration guide
0.04	29.07.2016	Christian KOCH	Changes according to comments
0.05	11.08.2016	Christian KOCH	Changes according to comments
1.00	11.08.2016	Cosmin BACIU	Describe how perform the validation of the submission. First version published
1.01	15.09.2016	Cosmin BACIU	Updated the document for 3.2-RC1
1.02	04.10.2016	Ioana DRAGUSANU	Added authentication details for 3.2.0
1.03	10.10.2016	Adrien FERIAL	Finalization of the document
1.04	19.01.2017	Cosmin BACIU	Documented the new *messageReceiveFailed* method
1.05	22.03.2017	Cosmin BACIU	Upgrade the version to 3.2.3
1.06	10.04.2017	Cosmin BACIU	Upgrade the version to 3.2.4
1.07	22/05/2017	Yves ADAM	Upgrade the version to 3.2.5
1.08	20/06/2017	Ioana DRAGUSANU, Cosmin BACIU	Add 3.3-RC1 changes, ext-services, domibus-logging, getStatus API
1.09	03/10/2017	Cosmin BACIU	Documented the new messageStatusChanged method
2.00	05/10/2014	Caroline AEBY	Document review and corrections.
2.01	09/10/2017	CEF Support	List of reviewers updated.
2.02	24/11/2017	CEF Support	Update with Domibus 3.3.1
2.03	08/02/2018	CEF Support	Update with Domibus 3.3.2
2.04	20/03/2018	CEF Support	Reuse notice added, update with Domibus 3.3.3
2.05	16/04/2018	CEF Support	Update with Domibus 3.3.4
2.06	01/08/2018	Caroline AEBY	Updated for Domibus 4.0. Added info on notification listener. Plugin services documented.
2.07	04/09/2018	Caroline AEBY	Removed references to release candidate version
2.08	07/09/2018	Cosmin BACIU	Updated for multi-tenancy
2.09	14/09/2008	Chaouki BERRAH	Minor Update from Ioana DRAGUSANU
2.10	26/09/2018	Caroline AEBY	Contact information update
2.20	27/11/2018	Caroline AEBY	Domibus 4.0.1 update

2.3	11/02/2019	Caroline AEBY	Domibus 4.0.2 update
2.4	16/04/2019	Caroline AEBY	Domibus 4.1 updates
2.5	17/04/2019	Cosmin BACIU	Domibus 4.1 updates: added payloadSubmittedEvent & payloadProcessedEvent
2.6	07/05/2019	Cosmin BACIU	Domibus 4.1-RC1 updates
2.7	15/07/2019	Caroline AEBY	4.1-RC1 => 4.1
2.8	09/09/2019	Caroline AEBY	4.1.1 + reference to the SAD for Multitenancy
2.9	11/10/2019	Ioana Dragusanu	Domain handling in Multitenancy environment.
3.0	05/11/2019	Cosmin Baciu	Password encryption documented
3.1	06/11/2019	Cosmin Baciu	Notification listener information added
3.2	04/02/2020	Caroline AEBY	Domibus 4.1.3
3.3	14/04/2020	Caroline AEBY	Remark on common claassloader and spring context between standard and custom plugins

Table of Contents

1. INTRODUCTION

This document describes the Domibus plugin architecture and plugin API.

1.1. Purpose

After reading this document, the reader should be aware of the capabilities provided by the Domibus plugin system. Additionally, a developer familiar with the AS4 protocol will be able to implement a plugin integrating an existing back office application into Domibus.

1.2. Users

This document is intended for the Directorate Generals and Services of the European Commission, Member States (MS) and companies of the private sector wanting to set up a connection between their backend system and the Access Point.

In particular:

- Business Architects will find it useful for determining how to best exploit the Access Point to create a fully-fledged solution.
- Analysts will find it useful to understand the Use-Cases of the Access Point.
- Architects will find it useful as a starting point for connecting a Back-Office system to the Access Point.
- Developers will find it essential as a basis of their development concerning the Access Point services.
- Testers can use this document in order to test the use cases described.

2. Backend Integration

2.1. General Overview

The purpose of Domibus is to facilitate B2B communication. To achieve this goal it provides a very flexible plugin model which allows the integration with nearly all back office applications.

There are three default plugins available with the Domibus distribution:

- the domibus-default-jms-plugin
- the domibus-default-ws-plugin
- the domibus-default-fs-plugin

Further documentation about those plugins can be found at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus .

Important remark: Developers of custom plugins should ensure that their plugins can be deployed alongside the standard Domibus Default Plugins, given that they share a common classloader and Spring context.

2.2. Plugin Structure

A plugin is dependent on the ***domibus-plugin-api*** module which is released together with the main Domibus application. Any changes to previous API versions will be addressed in a migration guide.

In addition to this required module, another module is available (more information about this module can be found in the **Domibus Software Architecture Document**[1]):

- ***domibus-logging*****:** may be used to maintain a uniform logging style with the Domibus core.

A plugin consists of the implementation of at least two interfaces, eu.domibus.plugin.transformer.MessageSubmissionTransformer and eu.domibus.plugin.transformer.MessageRetrievalTransformer, and the extension of one abstract class, eu.domibus.plugin.AbstractBackendConnector.

This way multiple plugins can share the same data formats while using different transport protocols or enforcing different security policies. It is also possible to implement transport handlers for protocols while keeping the actual data format pluggable as those classes are not necessarily coupled and can be reused independently from each other.

[1] The document can be downloaded at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus in the documentation section.

2.3. Message Flow

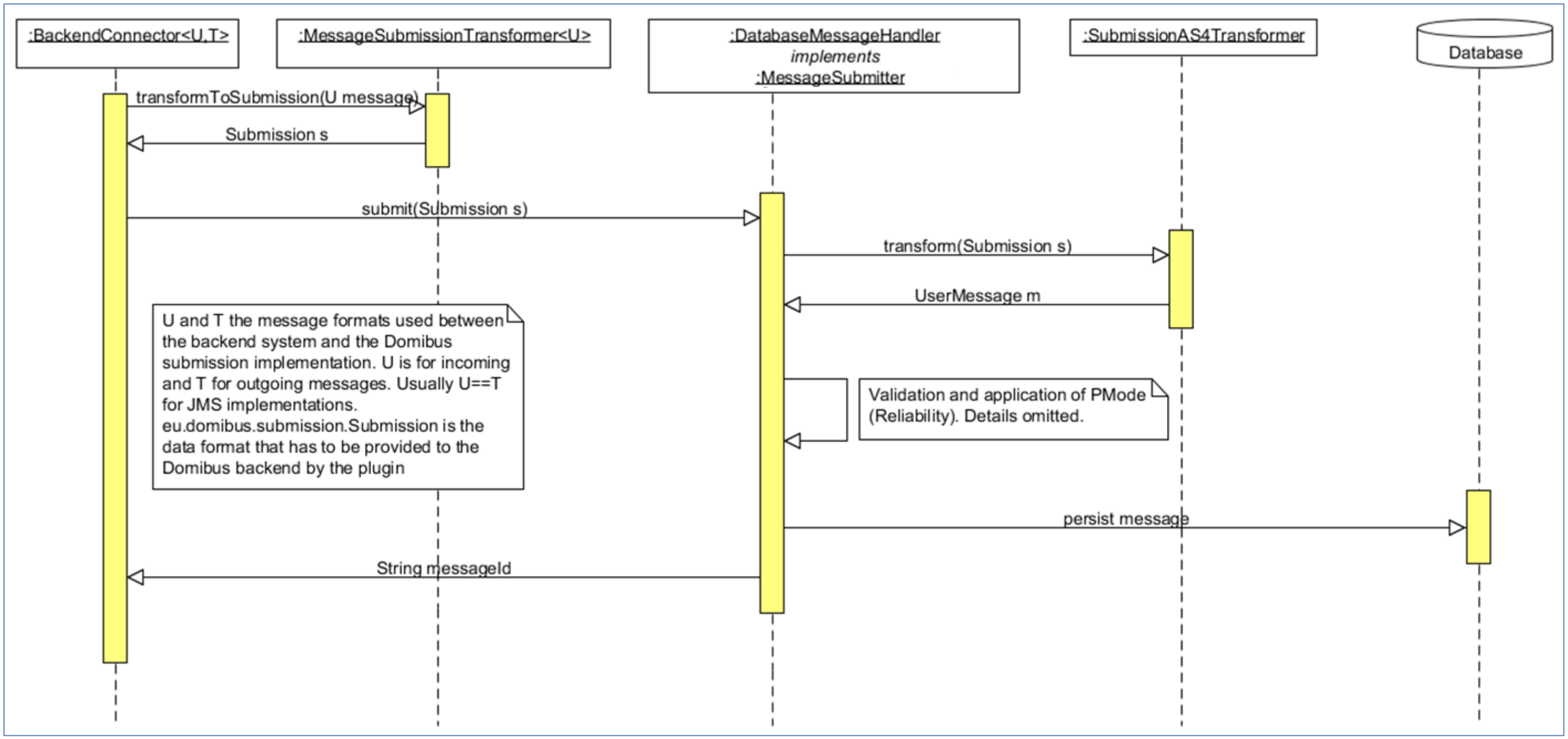

Figure 1 - Message Submission from the backend

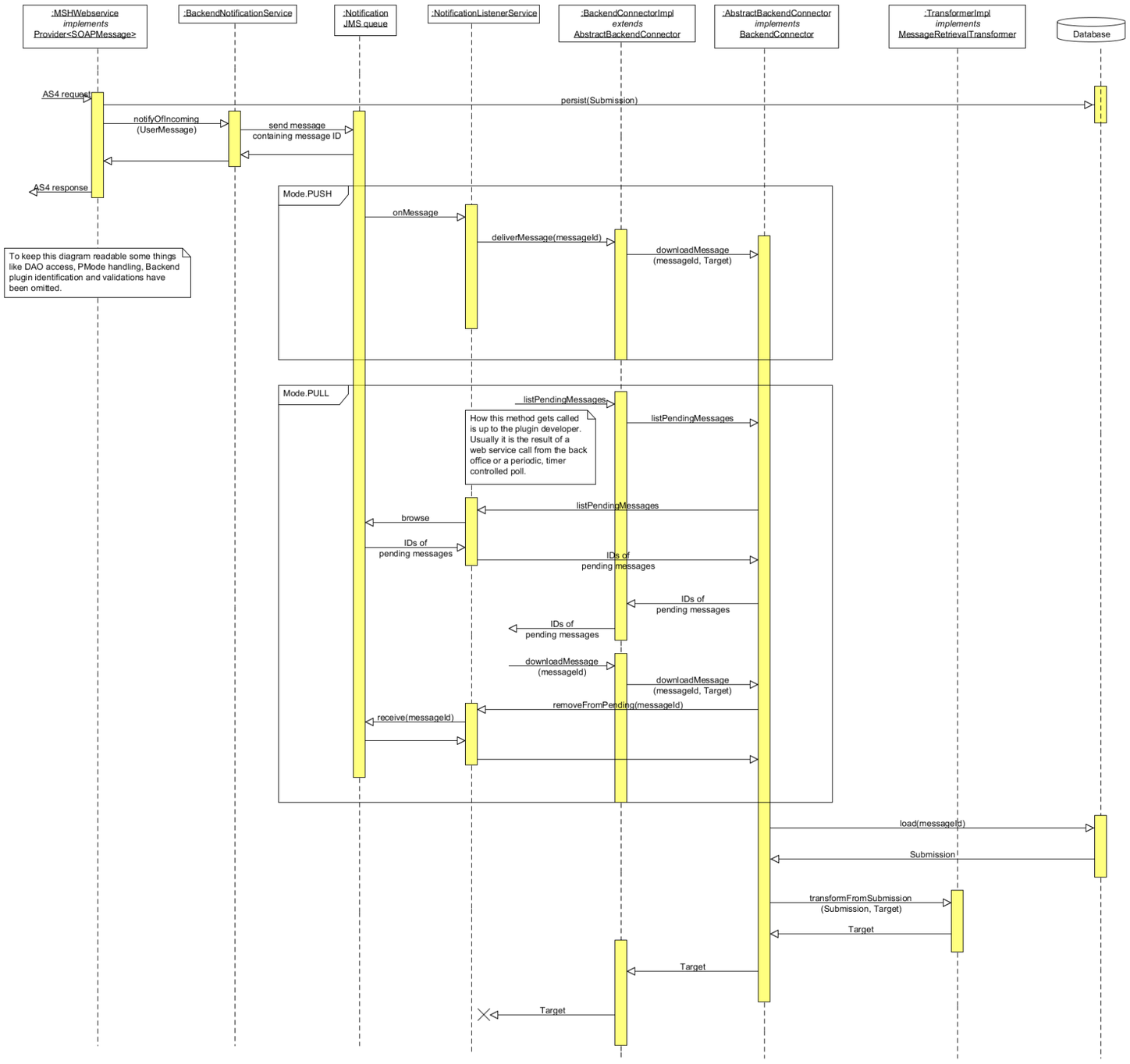

Figure 2 - Message reception by the backend and delivery to the plugin (PUSH/PULL mode)

3. Implementing a Plugin

3.1. Pull and Push plugins

There are two different ways of communicating with the backend. The first one is *eu.domibus.plugin.BackendConnector.Mode.PULL*. A plugin operation under this mode initiates all communications with *Domibus MSH* by itself and is never called from *Domibus* itself. This mode is intended mostly for backend applications that are not always online (e.g. mobile devices).

The default *Webservice* plugin bundled with *Domibus* is an example of such a plugin. The major disadvantage of this mode is that there is no way for *Domibus* to report processing errors back to the plugin, relying on calls to *getStatus(java.lang.String)* by the backend to be informed about any error conditions.

The preferred way of implementing a plugin is *eu.domibus.plugin.BackendConnector.Mode.PUSH*. This mode allows Domibus to push notifications and incoming messages to the backend. In case the backend is not reachable, the notification will be retried according to the backend queue definition as described in Section B.3, “Plugin configuration and deployment”. The default bundled JMS plugin is an example of such a plugin.

3.2. Extending eu.domibus.plugin.AbstractBackendConnector

eu.domibus.plugin.AbstractBackendConnector provides implementations of most methods defined in *eu.domibus.plugin.BackendConnector. eu.domibus.plugin.AbstractBackendConnector* should be used as basis for every plugin.

3.2.1. *eu.domibus.plugin.BackendConnector.Mode.PULL*

- getMessageSubmissionTransformer()
- getMessageRetrievalTransformer()
- payloadSubmittedEvent (eu.domibus.common.PayloadSubmittedEvent): Notifies the plugins for every payload that has been submitted to C2 but not yet saved
- payloadProcessedEvent (eu.domibus.common.PayloadProcessedEvent): Notifies the plugins for every payload that has been saved by C2

To submit a message to the MSH the *submit(U)* implementation of *eu.domibus.plugin.AbstractBackendConnector* should be used.

To download a message a combination of *listPendingMessages()* and *downloadMessage(java.lang.String, T)* should be used.

3.2.2. *eu.domibus.plugin.BackendConnector.Mode.PUSH*

- getMessageSubmissionTransformer()
- getMessageRetrievalTransformer()
- deliverMessage(java.lang.String)
- messageReceiveFailed(eu.domibus.common.MessageReceiveFailureEvent)

- messageSendFailed(java.lang.String)
- messageSendSuccess(java.lang.String)
- messageStatusChanged(eu.domibus.common.MessageStatusChangeEvent event)
- payloadSubmittedEvent (eu.domibus.common.PayloadSubmittedEvent): Notifies the plugins for every payload that has been submitted to C2 but not yet saved
- payloadProcessedEvent (eu.domibus.common.PayloadProcessedEvent): Notifies the plugins for every payload that has been saved by C2

Additionally *listPendingMessages()* is only callable from *Mode.PULL* plugins. To submit a message to the MSH the *submit(U)* implementation of *eu.domibus.plugin.AbstractBackendConnector* should be used. Additional details on these methods can be found in the Javadoc (see chapter 5).

3.3. Implementing eu.domibus.plugin.transformer.MessageSubmissionTransformer and eu.domibus.plugin.transformer.MessageRetrievalTransformer

The implementations of the transformer classes are responsible for transformation between the native backend formats and *eu.domibus.plugin.Submission*. As there are two different interfaces to implement it is possible to use different DTOs for message submission and reception. This is convenient when those tasks are handled by different backend applications.

As *eu.domibus.plugin.Submission* is able to represent all kinds of messages there are many parameters that must be set, with some of them unknown to the backend application. One approach is to statically set those values in the transformer classes. Another, more flexible approach is the usage of overridable default settings as used in the bundled default JMS plugin. For further details, see the documentation and implementation of the default JMS plugin.

3.4. Notification Listener

Domibus sends notifications to the plugins when different message events occur. The plugin can be notified via the backend notification queue or directly via callbacks. More information about which callbacks are available can be found *eu.domibus.plugin.NotificationListener* interface.

For example, the WebService plugin receives notifications on the *jms/domibus.notification.webservice* queue.

The Notification Types are:

```
public enum NotificationType {
   MESSAGE_RECEIVED, MESSAGE_FRAGMENT_RECEIVED, MESSAGE_SEND_FAILURE,
MESSAGE_FRAGMENT_SEND_FAILURE, MESSAGE_RECEIVED_FAILURE,
MESSAGE_FRAGMENT_RECEIVED_FAILURE, MESSAGE_SEND_SUCCESS,
MESSAGE_FRAGMENT_SEND_SUCCESS, MESSAGE_STATUS_CHANGE,
MESSAGE_FRAGMENT_STATUS_CHANGE;
}
```

Each plugin may configure its own list of notification types for which it expects to be notified. This list is optional. By default, plugins that use PULL mode receive notifications for ***MESSAGE_RECEIVED***, ***MESSAGE_SEND_FAILURE***, ***MESSAGE_RECEIVED_FAILURE*** while the PUSH plugins receive notification for all.

This list is passed as a constructor to the notification listener bean, in the *-plugin.xml file.

Example:

```
<util:list id="requiredNotificationsList" value-type="eu.domibus.common.NotificationType">
    <value>MESSAGE_RECEIVED</value>
    <value>MESSAGE_SEND_FAILURE</value>
    <value>MESSAGE_STATUS_CHANGE</value>
</util:list>

<bean id="webserviceNotificationListenerService"
    class="eu.domibus.plugin.NotificationListenerService"
    c:queue-ref="notifyBackendWebServiceQueue" c:mode="PULL"
    p:backendConnector-ref="backendWebservice"/>
    p:backendConnector-ref="backendWebservice">
    <constructor-arg ref="requiredNotificationsList"/>
</bean>
```

3.5. Validation of the submission

There are uses cases when it is required that the Submission object is validated before it is being delivered to the plugin. For instance, the user might want to verify that one of all the payloads is valid against a custom XSD schema. In this case, it does not make sense to deliver the message to the plugin for processing if it is not valid.

In order to better understand why the current API is not sufficient for this use case we have to understand first how the Submission object is delivered to the plugin for processing.

There are two transactions involved in the Submission processing:

1. In the first transaction, the message is stored in the database and a signal is sent internally via JMS to trigger the Submission processing.

2. A JMS listener is listening to Submission processing events and triggers the processing.

If the Submission is validated in the second step it would be too late because the Submission has been already saved and accepted for processing in the first step. This is the reason why we need to perform the Submission validation in the first step. If the Submission is not valid, an exception will be raised and the processing will not be performed.

The API for Submission validation can be found in the plugin API under the package *eu.domibus.plugin.validation*.

Hereunder you can find the class diagram of the classes involved in the submission validation:

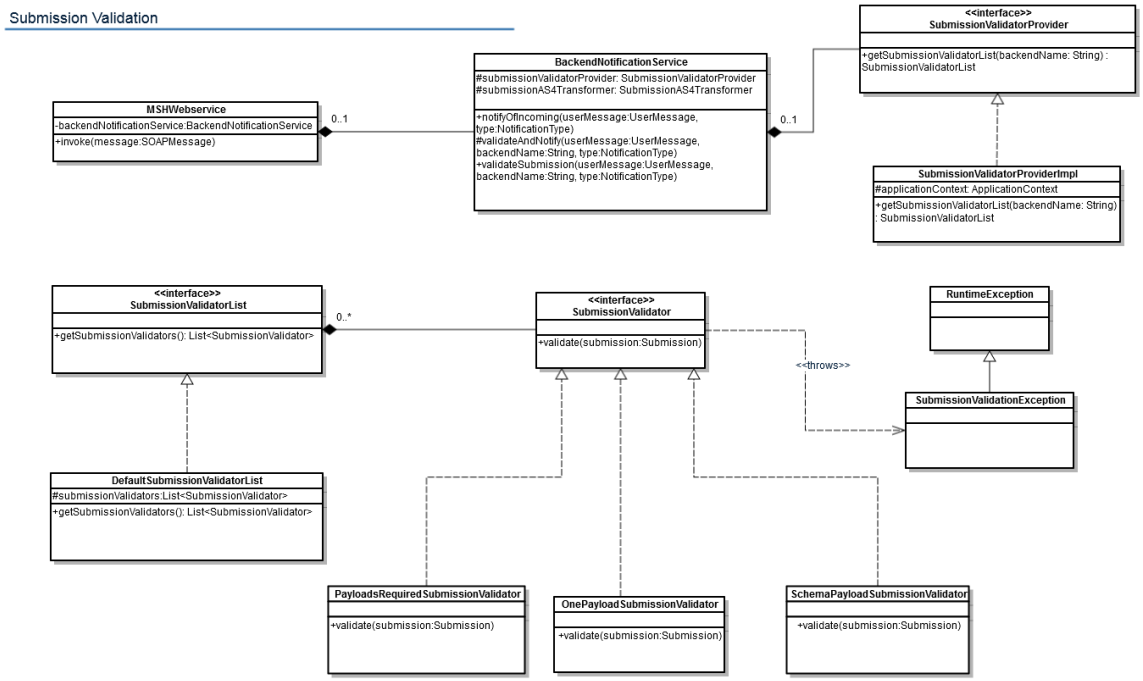

Figure 3 – Submission validation class diagram

In order to validate the Submission object, one has to declare in the plugin Spring context a bean of type *eu.domibus.plugin.validation.SubmissionValidatorList.* The bean id needs to contain the plugin name. The core will automatically discover the bean of type *SubmissionValidatorList* and perform the validation by calling the *validate* method on each configured *SubmissionValidator*.

In the plugin API there is already a default implementation of the *SubmissionValidatorList* interface *DefaultSubmissionValidatorList* that has an *java.util.ArrayList* for maintaining the list of validators.

By default Domibus comes with 3 implementations of the *SubmissionValidator* interface. An example how to use them can be found in the next paragraph.

1. *eu.domibus.submission.validation.OnePayloadSubmissionValidator* – validates that there is at least one payload present in the Submission

2. *eu.domibus.submission.validation.PayloadsRequiredSubmissionValidator* – validates that there is only one payload present in the Submission

3. *eu.domibus.submission.validation.SchemaPayloadSubmissionValidator* – validates that the payloads are valid against a custom XSD schema

Below is an extract of a custom plugin Spring context where we can see that a custom validator has been implemented and there are 3 validators used to validate the Submission:

```
<!-- custom validator -->
<bean id="customValidator"
class="eu.domibus.submission.validation.CustomSubmissionValidator"/>


<bean id="customJaxbContext" class="javax.xml.bind.JAXBContext" factory-
method="newInstance">
    <constructor-arg type="java.lang.String"
                     value="eu.domibus.plugin.custom.domain"/>
</bean>
<!-- schema validator -->
<bean id="customPayloadSchemaValidator"
class="eu.domibus.submission.validation.SchemaPayloadSubmissionValidator">
    <property name="jaxbContext" ref="customJaxbContext"/>
    <property name="schema" value="classpath:xsd/as4Payload.xsd"/>
</bean>

<!-- validators list -->
<bean id="customSubmissionValidatorList"
class="eu.domibus.plugin.validation.DefaultSubmissionValidatorList">
    <property name="submissionValidators">
        <list>
            <ref bean="onePayloadSubmissionValidator"/>
            <ref bean="customValidator"/>
            <ref bean="customPayloadSchemaValidator"/>
        </list>
    </property>
</bean>
```

3.6. Plugin Security

By default, the plugins security is disabled. Domibus can be configured to require authorization by setting the following property to **false** in the ***domibus.properties*** configuration file*:*

```
domibus.auth.unsecureLoginAllowed=false
```

Once the plugins security is activated, all the methods of the **eu.domibus.plugin.AbstractBackendConnector** class can only be called by authenticated users.

3.6.1. Authentication

The service **eu.domibus.ext.services.AuthenticationExtService** provided in the plugin API can be used by the plugins to authenticate the request.

It provides to the plugins two Java methods to authenticate:

1. **authenticate (HttpServletRequest httpRequest) throws AuthenticationExtException**

   This method supports the following authentication types:

   - Basic Authentication
   - X509Certificates Authentication
   - Blue Coat Authentication

Note: *Blue Coat is the name of the reverse proxy at the EC. It forwards the request in HTTP with the certificate details inside the request ("Client-Cert" header key).*

The **authenticate** method evaluates the three supported authentication methods in the following order: Basic Authentication, X509Certificates, Blue Coat certificates. The first authentication method found will execute, and the remaining authentication methods will not be evaluated anymore.

2. **basicAuthenticate(String username, String password) throws AuthenticationExtException;**

   This method supports only basic authentication.

All the users configured in the **Plugin User** UI page can authenticate and call any operation of the **eu.domibus.plugin.AbstractBackendConnector**.

By default there are two plugin users defined:

- "**admin**" has the role ROLE_ADMIN
- "**user"** has the role ROLE_USER, configured with Original User "**urn:oasis:names:tc:ebcore:partyid-type:unregistered:C1**"

Custom plugins may use their own custom authentication providers and perform different types of authentication. In case of custom authentication, the Spring **SecurityContextHolder** has to set correctly the **authentication** parameter after a successful authentication:

```
SecurityContextHolder.getContext().setAuthentication(authentication)
```

It is mandatory that the method **getPrincipal()** of the **authentication** parameter set above returns the original user value associated to the authenticated user. This original user value is used to authorize the user to a specific message. More information on how it is implemented can be found in **3.6.2 Authorization.**

3.6.2. Authorization

The authorization for the method defined in **eu.domibus.plugin.AbstractBackendConnector** is performed at Java method level using Spring **@PreAuthorize** annotation.

```
@PreAuthorize("hasAnyRole('ROLE_USER', 'ROLE_ADMIN')")
public void hasUserOrAdminRole() {}

@PreAuthorize("hasAnyRole('ROLE_ADMIN')")
public void hasAdminRole() {}
```

There are three roles defined for the plugin users **ROLE_AP_ADMIN, ROLE_ADMIN** and **ROLE_USER**, described below.

A user with role **ROLE_AP_ADMIN** or **ROLE_ADMIN** has the right to call the following methods of **eu.domibus.plugin.AbstractBackendConnector**:

- ***submit***
- ***downloadMessage***
- ***listPendingMessages***
- ***getStatus***
- ***getMessageErrors***

A user with role **ROLE_USER** associated to an Original User has the right to call the following methods of **eu.domibus.plugin.AbstractBackendConnector**:

- ***submit*** when the value of the **originalSender** from the submitted message is equal to the Original User of the authenticated user
- ***downloadMessage***, only if the **finalRecipient** value from the message to be downloaded is equal the Original User of the authenticated user
- ***listPendingMessages***, pending messages for which the **finalRecipient** value is equal to the Original User of the authenticated user
- ***getStatus*** and getMessageErrors when the value of the **originalSender** or **finalRecipient** of the message is equal to the Original User of the authenticated user**.**

3.7. Logging

The logging service is provided in the ***domibus-logging*** module, which is released together with the main Domibus application. More information about ***domibus-logging*** module can be found in the **Domibus Software Architecture Document** (the document can be downloaded at https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus in the documentation section).

Example of use:

private static final DomibusLogger LOG = DomibusLoggerFactory.getLogger(BackendWebServiceImpl.class);

3.8. Plugin Services

The Plugin API offers several services like monitoring or message acknowledgment, which are described below.

These services can be accessed in two ways:

- Java API

It can be used by the plugin implementers of the custom plugins.

- REST interface

The REST interface can be used directly by the C1/C4 backends if the network configuration allows it.

The documentation of the REST interface can be found on CEF Digital site.

3.8.1. Message acknowledgement service

This service is used to acknowledge when a message is:

- delivered from C3 to the backend
- processed by the backend

Here are the typical use cases for using the *MessageAcknowledgementService*:

- a message is received by C3 from C2: the plugin that handles the message registers an acknowledgment before delivering the message to the backend
- a message is processed by the backend and it informs C3 via the plugin; the plugin registers an acknowledgment that the message has been processed by the backend
- a message is processed by the backend and informs C3 directly via the REST service exposed by the core; a REST service is exposed containing the same signature as {@link MessageAcknowledgeService}

There are two ways of performing message acknowledgments between C3 and the backend:

- synchronous

C3 (via the plugin) notifies the backend synchronously and the backend process the messages also synchronously. In this case, there is no need for the backend to send a separate message acknowledgement so the plugin at the C3 side registers the processing of the message by the backend.

```
Eg:
BackendResponse backendResponse = plugin.callBackendWS(message)
messageAcknowledgeService.acknowledgeMessageDelivered(message.getId(), new
Timestamp(System.currentTimeMillis()))
messageAcknowledgeService.acknowledgeMessageProcessed(message.getId(), new
Timestamp(System.currentTimeMillis()))
```

- asynchronous

C3 notifies the backend synchronously and the backend process the messages asynchronously. In this case, the backend will send a separate message acknowledgement when it manages to process the message successfully.

```
Eg:
plugin.sendMessageToTheBackend(message)
messageAcknowledgeService.acknowledgeMessageDelivered(message.getId(), new
Timestamp(System.currentTimeMillis()))
```

3.8.2. Monitoring service

This service is used to monitor failed messages and to restore them if necessary.

Assuming that "failed message" means failed to be sent by the sender access point and getting the status set to SEND_FAILURE, the service gives the possibility to:

- list all the failed messages
- restore a failed message
- restore all messages failed during a specific period
- know how long time a message has been failed
- get the history of all delivery attempts
- delete the message payload of a failed message

3.9. Password encryption

Passwords configured in the plugin configuration files are stored by default in clear text.

Domibus gives the possibility to the plugins to encrypt passwords configured in the plugin configuration file using symmetric encryption with *AES/GCM/NoPadding* algorithm.

The *eu.domibus.plugin.encryption.PluginPropertyEncryptionListenerinterface* interface should be implemented by the custom plugins that would like to support password encryption. If password encryption is active (*domibus.password.encryption.active=true*), Domibus will generate the secret keys that are used to encrypt the passwords. It will notify afterwards via the *PluginPropertyEncryptionListenerinterface* interface the subscribed plugin listeners to encrypt their own passwords. Custom plugins can use the password encryption services, *eu.domibus.ext.services.PasswordEncryptionExtService*, to handle the password encryption.

The password encryption has been implemented in the Default File System Plugin, which can serve as an example on how to implement password encryption in your own custom plugin.

For instance, the property *fsplugin.authentication.password=test123* will be encrypted to *fsplugin.authentication.password=ENC(4DTXnc9zUuYqB0P/q7RtRHpG9VJLs3E=).*

4. PLUGIN CONFIGURATION AND DEPLOYMENT

The documentation for configuration of the message routing and plugin deployment for all supported deployment platforms can be found in the administration guide. It can be downloaded from the release page of Domibus, section Documentation:
https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus

5. API Documentation

Standard Javadoc documentation for the API can be downloaded on the https://ec.europa.eu/cefdigital/wiki/display/CEFDIGITAL/Domibus.

This documentation includes all necessary information required to implement the necessary methods.

6. MULTITENANCY

To understand how Multitenancy works in Domibus, please refer to Domibus Software Architecture Document (SAD), chapter “Multitenancy” and especially in the section “Plugins”.

After reading the SAD, the following aspects should be clear in a multitenant environment:

- Authentication is always required to submit a message (C1 to C2),
- Every plugin user is associated to one and only one domain,
- Every domain has its own schema in the database,
- A general schema exists to match users to domains,
- Default plugins already work in multitenant environment; they can be used as examples.

Any custom plugin should work in a Domibus multitenant environment.

Domibus Plugin API offers the possibility to handle the domains and access the database schema specific to each domain. Examples of domain handling using the Domibus plugin API are presented below.

6.1.1. Check if Domibus runs in Multitenancy mode:

```
@Autowired
protected DomibusConfigurationExtService domibusConfigurationExtService;

boolean isMultiTenantAware = domibusConfigurationExtService.isMultiTenantAware();
```

6.1.2. Get the current domain:

```
@Autowired
private DomainContextExtService domainContextExtService;

DomainDTO domainDTO = domainContextExtService.getCurrentDomainSafely();
```

6.1.3. Set the current domain for an asynchronous execution:

When the plugin uses asynchronous execution, the domain code is required to be configured manually.

Let’s take the scenario where a message is put in a JMS queue and a worker picks it up and sends it.

Along with the message, the domain code must be added in the JMS message. The consumer should first get the domain based on the domain code, set it as current domain, and afterwards execute the job.

Finally, the plugin is responsible to clear the current domain.

Get the domain:

```
@Autowired
protected DomainExtService domainExtService;

final DomainDTO = domainExtService.getDomain(domainCode)
```

Set current domain, execute job and clear domain:

```
@Autowired
protected DomainContextExtService domainContextExtService;

try {
   domainContextExtService.setCurrentDomain(currentDomain);

   // … executeJob

} finally {
   domainContextExtService.clearCurrentDomain();
}
```

6.1.4. Access the database:

Every domain has its own schema in the database. In case the plugin needs a new table to persist plugin specific data, it is recommended to keep the same segregation as the Domibus core and add a new table in the database schema of each domain.

The recommended naming convention is **<plugin_name>_TB_<name>** for tables storing plugin data and **<plugin_name>_QRTZ_<name>** for quartz tables.

See section §6.1.5 to see how to create a quartz job.

Below example shows how to save message info in a new table:

a) Create the tables in the database (to be performed by the database admin)

Table: WS_PLUGIN_TB_MESSAGE

- ID_PK - primary key (auto increment)
- DESCRIPTION

b) Create the entity class:

```
import javax.persistence.*;

@Entity
```

```
@Table(name = "WS_PLUGIN_TB_MESSAGE")
public class MessageInfoEntity {

   @Id
   @Column(name = "ID_PK")
   private long entityId;

   @Column(name = "DESCRIPTION")
   private String description;

   /**
    * @return the primary key of the entity
    */
   public long getEntityId() {
      return this.entityId;
   }

   public void setEntityId(long entityId) {
      this.entityId = entityId;
   }

   public MessageInfoEntity(final String description){
      this.description = description;
   }

   public String getDescription() {
      return description;
   }

   public void setDescription(String description) {
      this.description = description;
   }
}
```

c) Create a basic DAO implementation providing the standard CRUD operations. The class reuses the Domibus EntityManager.

```
public abstract class BasicDao<T> {

   protected final Class<T> typeOfT;

   @PersistenceContext(unitName = "domibusJTA")
   protected EntityManager em;

   /**
    * @param typeOfT The entity class this DAO provides access to
    */
   public BasicDao(final Class<T> typeOfT) {
      this.typeOfT = typeOfT;
   }
```

```
    public <T> T findById(Class<T> typeOfT, String id) {
        return em.find(typeOfT, id);
    }

    @Transactional(propagation = Propagation.REQUIRED)
    public void create(final T entity) {
        em.persist(entity);
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void delete(final T entity) {
        em.remove(em.merge(entity));
    }

    public T read(final long id) {
        return em.find(this.typeOfT, id);
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void updateAll(final Collection<T> update) {
        for (final T t : update) {
            this.update(t);
        }
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void deleteAll(final Collection<T> delete) {
        for (final T t : delete) {
            this.delete(t);
        }
    }

    @Transactional(propagation = Propagation.MANDATORY)
    public void update(final T entity) {
        em.merge(entity);
    }

    public void flush() {
        em.flush();
    }
}
```

Note that the EntityManager is already configured in Domibus core to be multitenant aware and persist the data in the right table, of the current domain.

d) Create the MessageInfoDao class, that extends the BasicDao:

```
@Repository
public class MessageInfoDao extends BasicDao<MessageInfoEntity> {

   public MessageInfoDao() {
      super(MessageInfoEntity.class);
   }
}
```

e) Save the message info:

```
@Autowired
MessageInfoDao messageInfoDao;

String description = "New message messageId = " + messageInfo.getMessageId();
messageInfoDao.create(new MessageInfoEntity(description));
```

6.1.5. Create a quartz job:

To create a new Quartz job that executes plugin specific tasks, it is recommended to extend the abstract class DomibusQuartzJobExtBean and override the method executeJob() with the specific actions. By doing this, the domain configuration is handled internally by Domibus.

```
@DisallowConcurrentExecution // Only one Worker runs at any time on the same node
public class MyPluginWorker extends DomibusQuartzJobExtBean {

   @Override
   protected void executeJob(JobExecutionContext context, DomainDTO domain) {
      //do plugin specific tasks
   }
}
```

Create the quartz job:

```
<bean id="myPluginWorkerJob"
    class="org.springframework.scheduling.quartz.JobDetailFactoryBean">
   <property name="jobClass" value="eu.domibus.plugin.worker. MyPluginWorker "/>
   <property name="durability" value="true"/>
</bean>
```

Create the Quartz job trigger:

```
<bean id="MyPluginWorkerTrigger"
    class="org.springframework.scheduling.quartz.CronTriggerFactoryBean"
scope="prototype">
  <property name="jobDetail" ref=" myPluginWorkerJob "/>
  <property name="cronExpression" value=" 0 0/1 * * * ?"/>
  <property name="startDelay" value="20000"/>
</bean>
```

The default FileSystem plugin provides examples of how to create new Quartz jobs to purge sent, received and failed files that were archived.

7. LIST OF FIGURES

8. CONTACT INFORMATION

CEF Support Team

By email: CEF-EDELIVERY-SUPPORT@ec.europa.eu

Support Service: 8am to 6pm (Normal EC working Days)